420-SF2-RE — Structure de données et programmation orientée objet — Norme d'utilisation de UML

Par Godefroy Borduas — Hiver 2026

Objectif : Créer un standard d'utilisation du langage UML pour les cours 420 de SIM.

Structure du document

Les présents documents formalisent les règles d'utilisation du langage UML pour les cours 420 du programme de Sciences, informatique et mathématiques du Cégep du Vieux Montréal. Les règles qui suivent découlent de la norme UML 2.5. Néanmoins, le document cherche à standardiser les pratiques moins présentes dans la norme actuelle. L'objectif final est d'avoir une lecture commune du langage de conception.

Le document sera divisé en plusieurs sections qui représentent un aspect particulier ou un élément de la Programmation orientée objet. À chaque section, un diagramme de classe illustra l'idée à l'aide d'une illustration et du code PlantUML[1] qui le génère.[2]

Description des classes

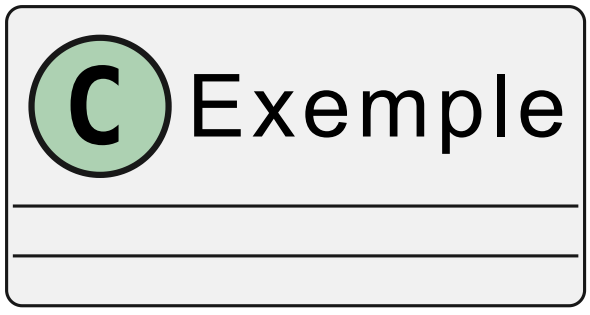
Définition d'une classe

Toutes les classes sont représentées par un rectangle subdivisé en trois sections. La première section contient le **nom de la classe**. Le nom sera accompagné des stéréotypes au besoin au-dessus de ce dernier.

La seconde section contiendra les **attributs de l'objet**. Chaque attribut sera composé de son modificateur d'accès, de son nom et de son type. Il est accompagné, lorsque nécessaire, de ces stéréotypes ou de ses propriétés (comme `static` ou `abstract`).

La dernière section contiendra les **méthodes de l'objet**. Chaque méthode era composée de son modificateur d'accès, de son nom et de son type. Il est accompagné, lorsque nécessaire, de ces stéréotypes ou de ses propriétés (comme `static` ou `abstract`).

> Les cercles avec une lettre, situé à gauche des noms de classe ou d'énumération, ne sont pas standard en UML. Il s'agit d'un ajout de PlantUML et d'autres logiciels.



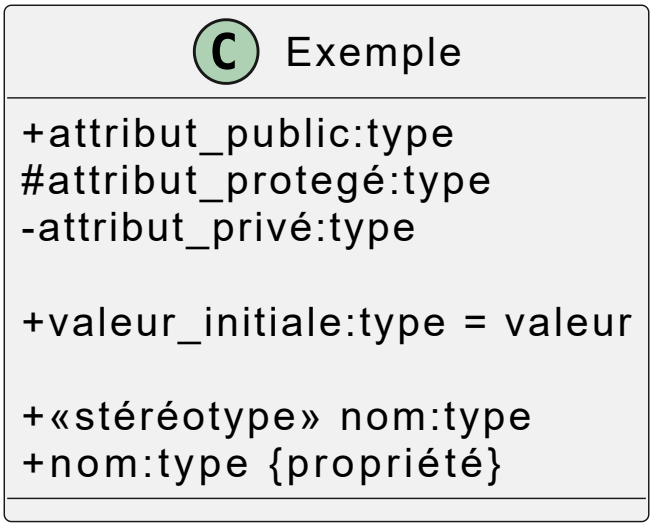
```
class Exemple {

}
```

Définition d'un attribut

Les attributs, situés dans la deuxième section des classes, sont représentés sur une seule ligne. Ils commenceront par le **modificateur d'accès** suivi des stéréotypes, si besoin, et du nom de l'attribut.

Après le nom, vous trouverez le type de l'attribut précédé du symbole `:`. Enfin, vous trouverez les propriétés de l'attribut.

Au besoin, vous pouvez définir, après le type et avant les propriétés, la valeur initiale de l'attribut. La valeur initiale sera précédée du symbole `=`.




```
skinparam classAttributeIconSize 0

class Exemple {
    + attribut_public:type
    # attribut_protegé:type
    - attribut_privé:type

    + valeur_initiale:type = valeur

    + <<stéréotype>> nom:type
```

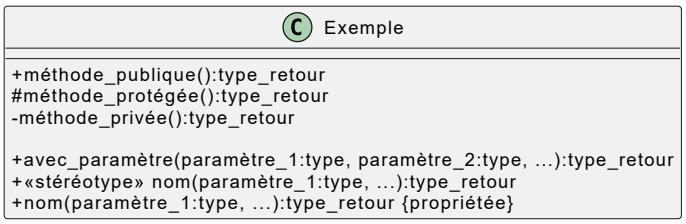
```
    + nom:type {propriété}
}
```

Définition d'une méthode

Les méthodes, situées dans la troisième section des classes, sont représentées sur une seule ligne. Ils commenceront par le **modificateur d'accès** suivi des stéréotypes, si besoin, et du nom de l'attribut.

Après le nom, vous trouverez les paramètres de la fonction. L'écriture des paramètres suivent la même syntaxe que les attributs : `nom:type [= valeur_par_défaut]`. Chaque paramètre est séparé par des virgules.

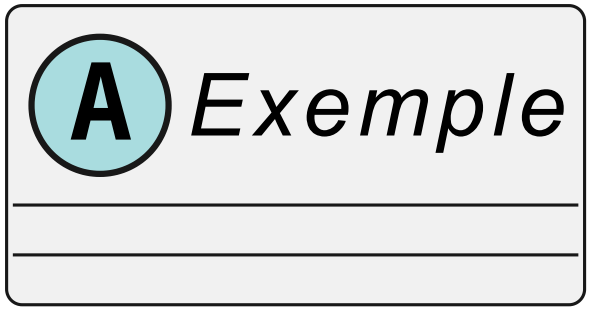
Après les paramètres, vous trouverez le type de retour de la méthode précédé du symbole `:`. Enfin, vous trouverez les propriétés de l'attribut.




```
skinparam classAttributeIconSize 0

class Exemple {
    + méthode_publique():type_retour
    # méthode_protégée():type_retour
    - méthode_privée():type_retour

    + avec_paramètre(paramètre_1:type, paramètre_2:type, ...):type_retour
    + <<stéréotype>> nom(paramètre_1:type, ...):type_retour
    + nom(paramètre_1:type, ...):type_retour {propriétée}
}
```

Définition d'une classe *abstraite*

Les classes abstraites sont définies selon les mêmes normes que les classes. Toutefois, la classe sera représentée par la lettre `A` et le nom sera en italique. De plus, cette règles s'applique à toutes les composantes abstraites d'un objet (méthode, propriété, etc.)

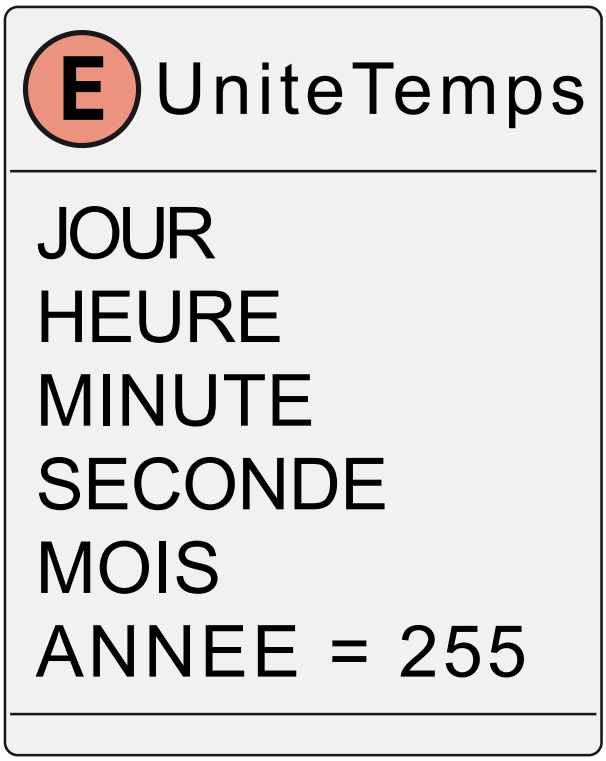



```
abstract Exemple {

}
```

Définition d'une énumération

Les énumérations sont définies selon les mêmes normes que les classes. Toutefois, la classe sera représentée par la lettre `E`.

De plus, les valeurs de l'énumération sont représentées dans la deuxième section. Chaque valeur sera composée de son nom et, si nécessaire, de sa valeur numérique précédés du symbole `=`. Il n'y a pas de modificateur d'accès pour les valeurs d'une énumération.



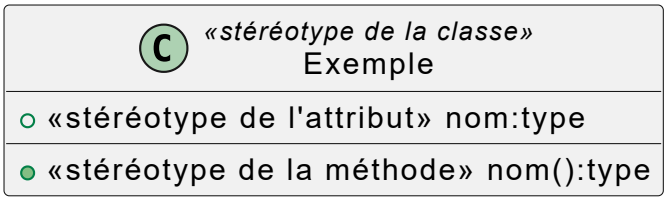

```
enum UniteTemps {
    JOUR
    HEURE
    MINUTE
    SECONDE
    MOIS
    ANNEE = 255
}
```

Spécifications des entités

Les stéréotypes

Autant pour les classes, les attributs ou les méthodes, les stéréotypes permettent d'étendre (au sens extensions) le langage UML. Ces extensions ajoute des éléments de notation à ceux de base (ex. `singleton`, `builder`, `property`, etc.) Ces extension permettent de spécifier des technicalité du domaine applicatif chez un client. Il s'agit d'élément propose à certain projet donné.

Les stéréotypes s'écrivent toujours entre chevrons `<< >>`.




```
class Exemple <<stéréotype de la classe>> {
    + <<stéréotype de l'attribut>> nom:type
    + <<stéréotype de la méthode>> nom():type
}
```

Les modificateurs d'accès

Aussi appelé la **visibilité**, il existe trois modificateurs d'accès en Python[3]. Il s'agit des modificateurs suivants : public, protégé (`protected`) et privé (`private`).

Le modificateur d'accès **public** est toujours représenté, dans le UML de base, par le symbole `+` . PlantUML utilise par défaut le cercle vert.

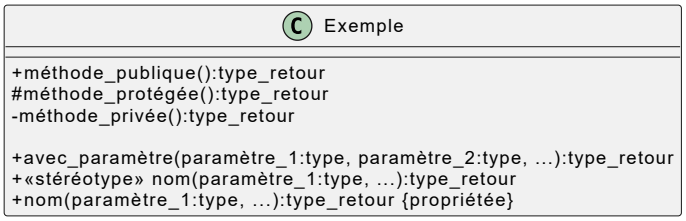
Le modificateur d'accès **protégé (`protected`)** est toujours représenté, dans le UML de base, par le symbole `#` . PlantUML utilise par défaut le losange jaune.

Le modificateur d'accès **privé (`private`)** est toujours représenté, dans le UML de base, par le symbole `-` . PlantUML utilise par défaut le carré rouge.

Lorsque vous réalisez un diagramme à la main, vous devez obligatoirement utiliser les **caractères de base**.

Lorsque vous utilisez PlantUML, vous pouvez utiliser les symboles de base ou les symboles du générateur.

Vous pouvez forcer PlantUML à utiliser les symboles de base avec l'instruction suivante : `skinparam classAttributeIconSize 0` .



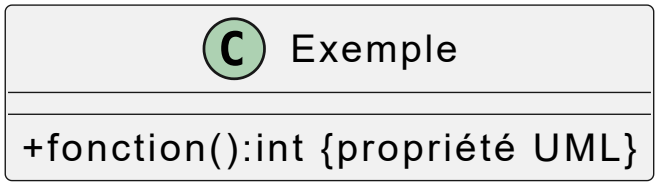
```
skinparam classAttributeIconSize 0

class Exemple {
    + méthode_publique():type_retour
    # méthode_protégée():type_retour
    - méthode_privée():type_retour

    + avec_paramètre(paramètre_1:type, paramètre_2:type, ...):type_retour
    + <<stéréotype>> nom(paramètre_1:type, ...):type_retour
    + nom(paramètre_1:type, ...):type_retour {propriétée}
}
```

Les propriétés UML

> **Attention !** Cette section est propre à Python et elle peut varier selon le langage utilisé.

Les propriétés UML précisent le comportement d'une entité (classe, attribut, méthode, etc.) Il s'agit d'un commentaire entre accolades `{ }` toujours situé à la fin de la ligne (de l'entité). En règle générale, le terme utilisé pour la propriété est aussi standardisé.

Petite précision, dans certains cas (comme la méthode statique ou abstraite), la propriété UML n'apparaîtra pas entre accolades. Il existe une autre symbolique liée.

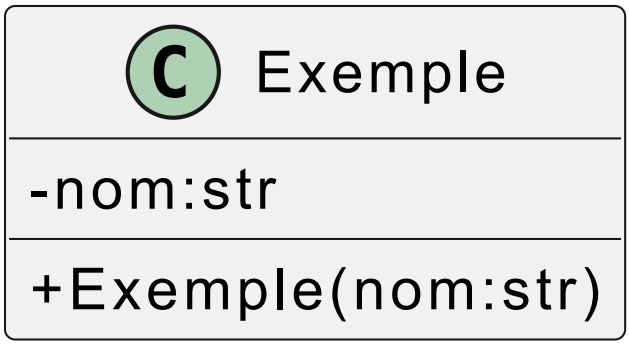



```
skinparam classAttributeIconSize 0

class Exemple {
    + fonction():int {propriété UML}
}
```

Précisions sur les classes

Définition du constructeur

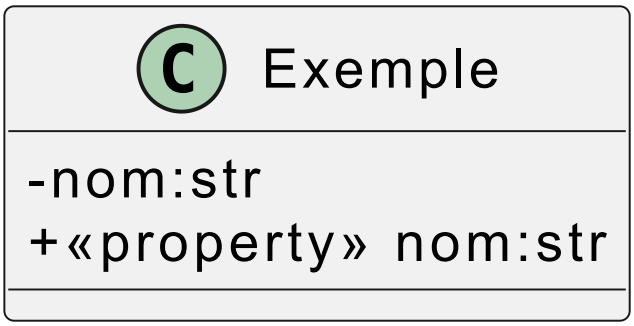
Les constructeurs sont des **méthodes *sans retour***. Contrairement à Python, leur nom correspond **toujours** au nom de la classe. Vous devez y définir l'ensemble des paramètres comme toutes autres méthodes.




```
skinparam classAttributeIconSize 0

class Exemple {
    - nom:str
    + Exemple(nom:str)
}
```

Définition des accesseurs (*getters*) et des mutateurs (*setters*)

Les propriétés sont des méthodes **écrites comme des attributs**. Il n'y a **jamais** de parenthèse ou de paramètre. Vous devez aussi utiliser le **stéréotype** `<<property>>` .

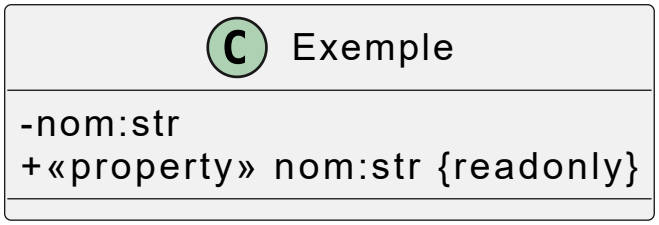



```
skinparam classAttributeIconSize 0

class Exemple {
    - nom:str
```

```
    + <<property>> nom:str
}
```

Définition des accesseurs (*getters*) en lecture seule

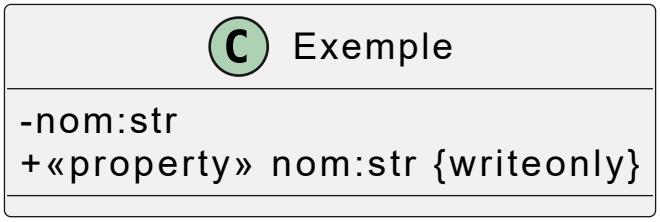
Lorsque l'accesseur est en lecture seule, vous devez toujours suivre la procédure sur les accesseurs et les mutateurs. Toutefois, les accesseurs seuls sont un cas particulier des `property`. Vous devez alors renseigner la propriété UML `{readonly}`.




```
skinparam classAttributeIconSize 0

class Exemple {
    - nom:str
    + <<property>> nom:str {readonly}
}
```

Définition des mutateurs (*setters*) en écriture seule

Lorsqu'un mutateur est en écriture seule[4], vous devez toujours suivre la procédure sur les accesseurs et les mutateurs. Toutefois, les accesseurs seuls sont un cas particulier des `property`. Vous devez alors renseigner la propriété UML `{writeonly}`.



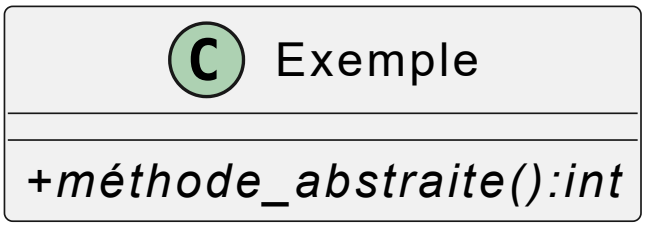

```
skinparam classAttributeIconSize 0

class Exemple {
```

```
    - nom:str
    + <<property>> nom:str {writeonly}
}
```

Déclaration de méthodes abstraites

Les méthodes abstraites sont déclarées comme toute autre méthode. Néanmoins, puisqu'il s'agit d'un cas particulier, vous devez renseigner la propriété UML `{abstract}` .

Les méthodes abstraites sont toujours affichées en italique. Lorsque vous réalisez un diagramme à la main, utilisez simplement la propriété UML.

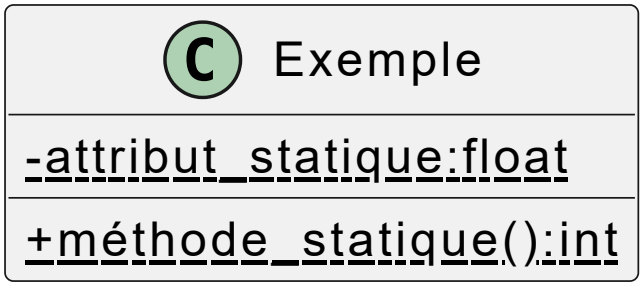



```
skinparam classAttributeIconSize 0

class Exemple {
    + méthode_abstraite():int {abstract}
}
```

Déclaration de méthodes statiques

Les méthodes statiques sont déclarées comme toute autre méthode. Néanmoins, puisqu'il s'agit d'un cas particulier, vous devez renseigner la propriété UML `{static}` .

Les méthodes statiques sont toujours soulignées. Lorsque vous réalisez un diagramme à la main, vous devez souligner vos méthodes.



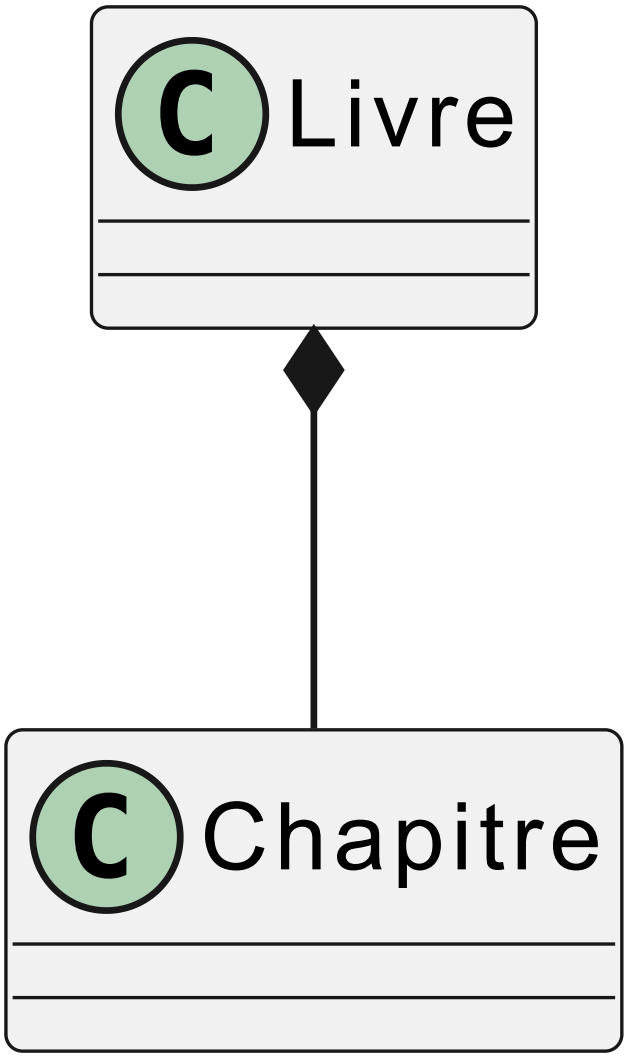

```
skinparam classAttributeIconSize 0

class Exemple {
    - attribut_statique:float {static}
    + méthode_statique():int {static}
}
```

Liens entre les classes

Composition

La composition est définie par un losange noir (♦) qui relie deux classes. Le losange est toujours fixé sur la classe composite, soit celle qui est composée de la classe partie. En PlantUML, le lien de composition est décrit par le symbole `*--` .

Vous devez toujours écrire le lien selon la syntaxe suivante : `Classe composite *-- Classe partie` .

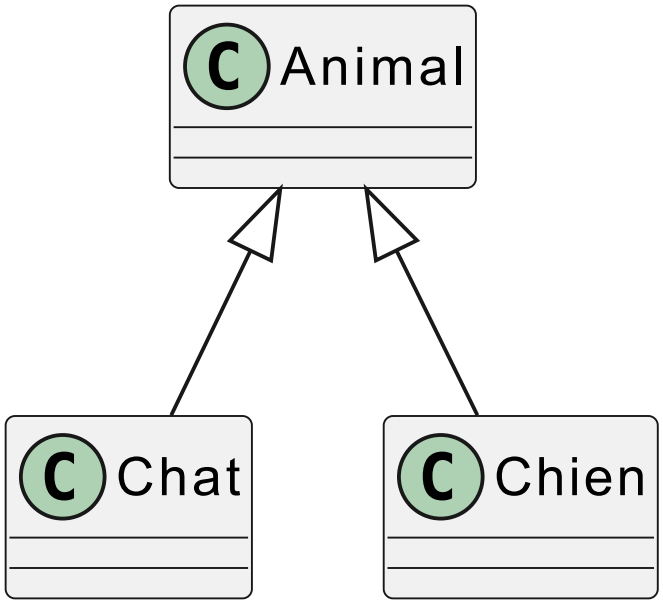



```
class Livre {
}

class Chapitre {

}

Livre *-- Chapitre
```

Héritage

La composition est définie par une flèche qui relie deux classes. La flèche est toujours fixée sur la classe parente. En PlantUML, le lien de composition est décrit par le symbole `^--` .

Vous devez toujours écrire le lien selon la syntaxe suivante : `Classe parente ^-- Classe enfant` .




```
class Animal {}

class Chat {}

class Chien {}

Animal ^-- Chat
Animal ^-- Chien
```

1. https://plantuml.com/fr/class-diagram - Langage de génération de schéma UML. ↩
2. https://www.uml-diagrams.org/class-reference.html - Référence de la norme ↩
3. Le modificateur d'accès `package private` n'existe pas en Python. ↩
4. Ce cas de figure est rare et particulier. Assurez-vous de sa nécessité avant de l'utiliser. ↩